Enhancing the Casper Library Security with Mandatory Access Control

1st Yan-Hao Wang
Computer Science and Information Engineering
National Central University
Taoyuan, Taiwan
bses30074@gmail.com

2nd Li-Wen Hsu
The FreeBSD Project
Taipei, Taiwan
lwhsu@FreeBSD.org

***Abstract*—Casper, a userspace library in FreeBSD, provides system services—such as DNS resolution—to applications confined by Capsicum. However, these Casper services execute with standard privileges, exposing the system to potential security risks, including unauthorized resource access. To mitigate this issue, we present a sandboxing mechanism for Casper based on FreeBSD's Mandatory Access Control (MAC) framework. Our design confines Casper services to MAC-enforced domains, thereby limiting their privileges and isolating their interactions with the system. We evaluate the proposed mechanism across multiple Casper services and show that it strengthens system security while incurring only modest performance overhead.**

I. Introduction

Casper [1] is a userspace service framework in FreeBSD that provides applications with access to system services—such as DNS resolution—while aiming to preserve Capsicum's security model. In the common Capsicum deployment pattern, an application is split into privileged and unprivileged processes; Casper helps these processes obtain necessary system services without granting the unprivileged part broad ambient authority.

However, Casper service processes themselves typically run with standard privileges, which introduces an additional attack surface. If a Casper service is compromised, it may be able to access resources beyond what the confined application would otherwise be allowed to reach.

Capsicum [2] is a capability-based sandbox framework for FreeBSD that extends the standard UNIX API with primitives for enforcing fine-grained security policies. While Capsicum effectively limits the privileges of sandboxed applications, it does not by itself constrain the privileges of the auxiliary Casper services they rely on.

This work proposes a Mandatory Access Control (MAC)-based approach to sandbox Casper and its associated service processes, thereby strengthening the security guarantees of the overall Capsicum-based architecture.

II. Related Works

A sandbox is a controlled execution environment that isolates a process, restricting its access to system resources [3]. The fundamental principle underlying sandboxing is least privilege [4], which ensures that each process operates with only the minimal set of permissions necessary to perform its intended function.

We can broadly categorize sandboxing approaches into two complementary perspectives [5]:

1) Subject-based sandboxing: These sandboxes primarily restrict the capabilities of the executing process (the subject), controlling what actions it can perform and which resources it can access.
2) Object-based sandboxing: These sandboxes focus on the resources or objects themselves, enforcing policies that dictate which subjects may interact with them and in what manner.

In practice, many systems adopt a hybrid approach that considers both the subject and the object, combining restrictions on the process with policies on the resources it accesses.

Sandboxing can be implemented at multiple layers of a system, with each layer providing complementary security guarantees:

- Hardware-level sandboxing: Leverages CPU or memory features to restrict access to specific memory regions or instructions. Examples include Intel Memory Protection Keys and ARM Memory Tagging Extension.
- Operating System-level sandboxing: Relies on the operating system's access control mechanisms to regulate process capabilities and resource access, ensuring policy-driven isolation between processes.

A. Linux seccomp and OpenBSD pledge/unveil

Linux provides `seccomp-bpf` [6], [7] to restrict the system calls available to a process. Seccomp allows a process to enter a "secure computing" mode in which only a defined set of system calls are permitted; all other calls are blocked or trigger a configurable action, such as process termination. The `seccomp-bpf` extension leverages Berkeley Packet Filter (BPF) programs to define fine-grained, programmable filters for system calls.

OpenBSD provides similar mechanisms with `pledge` [8] and `unveil` [9]. `pledge` allows a process to declare the set of operations it intends to perform, such as file I/O, networking, or process management. Any attempt to perform operations outside this set is immediately blocked by the kernel, providing a simple yet effective sandbox. `unveil` further restricts filesystem access by allowing a process to

reveal only specific paths to which it requires access. Together, these mechanisms enforce strict least-privilege principles at the process level and are often used in OpenBSD to confine applications.

B. FreeBSD Capsicum

While these mechanisms primarily target the single-process level, more complete isolation environments require combining multiple mechanisms. Docker, for example, leverages namespaces, cgroups, and seccomp-bpf to provide containerized isolation. Capsicum, by contrast, can provide a more complete isolation environment through a single mechanism.

Capsicum is a capability-based sandboxing framework that enhances process isolation by limiting the file descriptors a process can use. Capsicum operates on the principle of least privilege, restricting processes to only the resources they need to function. This approach mitigates the risk of privilege escalation and reduces the attack surface of applications. Capsicum has been integrated into the FreeBSD base system and is utilized in various applications to enforce fine-grained access control.

C. Access Control

Access control provides another approach to sandboxing by limiting access to system resources. Discretionary Access Control (DAC) was the first widely adopted model, focusing on user-oriented permissions. Unix implements DAC through user/group/other permissions and Access Control Lists.

Mandatory Access Control (MAC) provides a more fine-grained and policy-driven access control mechanism. Well-known implementations include Security-Enhanced Linux (SELinux) [10], AppArmor [11], and TOMOYO Linux on Linux systems, as well as the TrustedBSD MAC framework on FreeBSD [12]. These systems enforce security policies that restrict access based on labels or profiles, offering a higher level of security compared to DAC.

In this work, we adopt a MAC-based sandboxing approach, as it offers a practical balance between performance overhead and development complexity and is readily available on FreeBSD. This approach leverages the TrustedBSD MAC framework to enforce security policies that confine processes to the minimum resources necessary for their operation, thereby enhancing system security.

III. Problem Statement

While Capsicum provides a fine-grained capability model for restricting the privileges of application processes, it does not offer comprehensive support for operations that require asynchronous or stateful interaction with the kernel. In particular, network-related system calls remain outside its confinement model. For example, during a TCP `connect()`, the calling thread may block inside the kernel while awaiting an acknowledgment, enabling a window for post-condition attacks after arguments have already been validated [13]. This limitation necessitates the use of auxiliary privileged

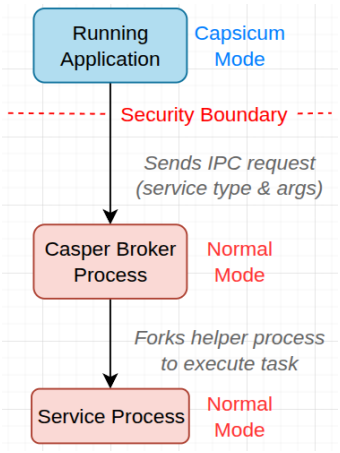

Fig. 1. Casper architecture and request flow.

```
system.dns      provides libc compatible DNS API
system.fileargs
                provides an API for opening files specified  on  a
                command line
system.grp      provides a getgrent(3) compatible API
system.net      provides a libc compatible network API
system.netdb    provides libc compatible network proto API
system.pwd      provides a getpwent(3) compatible API
system.sysctl   provides a sysctlbyname(3) compatible API
system.syslog   provides a syslog(3) compatible API
```

Fig. 2. Casper services provided by `libcasper(3)`.

components to perform certain system operations on behalf of sandboxed processes.

FreeBSD addresses this limitation by introducing Casper, a library that performs sensitive operations on behalf of Capsicum-restricted applications. As illustrated in Figure 1, Casper uses a centralized broker process and may fork service-specific helper processes to execute requested tasks. Figure 2 summarizes the system services provided by Casper, including DNS resolution, system database queries, and other network-related operations. This architecture allows applications to delegate privileged operations without breaking the object-capability model enforced by Capsicum.

However, this design introduces a critical security gap: both the Casper broker process and the service-specific helper processes execute with normal system privileges and are not themselves confined by Capsicum or any other isolation mechanism. Consequently, they become part of a large trusted computing base (TCB). If any Casper service is compromised, an attacker could gain unauthorized access to system resources, thereby undermining the security guarantees provided to the restricted application.

The core problem is the absence of a privilege-restriction or sandboxing mechanism for Casper and its service processes. Without additional containment, Capsicum deployments inherit a security boundary no stronger than their least secure Casper service. This motivates the need for a MAC-based confinement framework that isolates Casper within well-defined security domains while preserving its functionality and compatibility.

IV. Methodology

A. Privilege Analysis and Identification

To enforce the principle of least privilege effectively, it is crucial to accurately identify the permissions required by each Casper service.

FreeBSD provides DTrace [14], a comprehensive dynamic tracing framework for real-time monitoring of system calls and kernel interactions. We instrumented the Casper services with DTrace scripts to capture the privileges exercised during typical operations.

B. Least Privilege Policy Definition

Based on our analysis, we define a per-service least-privilege policy that preserves functionality while minimizing the attack surface.

Subject perspective. In brief, we constrain IPC, network, filesystem, and system interfaces to the minimum needed.

- **Communication:** Services are restricted to local IPC (UNIX domain sockets).
- **Network:** External connections are allowed only for network-facing services (e.g., DNS/NET).
- **Vnode:** File access is limited to the minimal configuration/state inputs; `fileargs` only delegates descriptors and is denied content I/O.
- **System:** Sensitive kernel interfaces (e.g., `sysctl`) are granted only to services that require them.

Object perspective. Table I lists the allowed file paths for each service; any other access is denied.

C. Enforcement via MAC Labels

We enforce the policy using the MAC framework by labeling both subjects (processes) and objects (vnodes):

- **Process Labels:** Assigned dynamically by the Casper broker when it forks a specific service process.
- **Vnode Labels:** Being persistent attributes of the file system, these must be assigned before any service execution.

To ensure consistent enforcement across reboots, we initialize required vnode labels at boot via an `rc.d` script.

D. Other Prohibited Privileges

Our MAC policy is deny-by-default: each Casper service is granted only the hooks required by its allowlist policy, and all other MAC hooks are denied. This ensures that compromised services remain confined to the intended least-privilege boundary.

E. Reproducibility (How to Use)

This subsection summarizes how to deploy our prototype and run Casper services under MAC confinement.

1) Prerequisites: We assume a FreeBSD environment with kernel/module build tooling and access to the source tree (commonly `/usr/src`). Administrator privileges are required. The file system should support MAC multilabel for persistent label storage.

2) Install and Configuration:

a. Fetch the project source code from the repository.
b. Enable MAC multilabel on the target file system (required for persistent labels).
c. Build and install the kernel MAC module, then configure it to load at boot (e.g., via `/boot/loader.conf`).
d. Install and enable the label initialization `rc.d` service so labels are applied on every boot.
e. Register the Casper label in `/etc/mac.conf` (append `"?casper"` to the `default_labels file` entry).
f. Patch and rebuild `libcasper`: replace `service.c` with the version described in the project README, then rebuild/install from `/usr/src/lib/libcasper`.

3) Run: After installation, Casper can be started and used as usual. Once Casper broker/services are launched, our MAC policy is applied automatically, and the services operate under restricted privileges.

V. Evaluation

A. Effectiveness Evaluation

To evaluate the security boundaries enforced by our MAC module, we developed an attack harness that simulates a compromised Casper service. The harness assumes that arbitrary code execution has been achieved and attempts to perform privileged operations defined in our threat model.

Table II maps attacker behaviors to the specific system calls triggered by the harness. We executed these attacks within a confined Casper service to verify that unauthorized operations are blocked at the kernel level.

We evaluated our MAC module using the `grp` service (`cap_grp(3)`) as the primary test target. This choice is strategic: while other services, such as `dns`, inherently require network capabilities, the `grp` service has a more restricted and deterministic functional scope. Using the `grp` service as a testbed allows us to establish a clean security baseline and clearly demonstrate that our MAC implementation intercepts and blocks unauthorized behaviors.

The list below details each attack scenario.

- **Process Control** (`ATTACK_EXEC`): Attempts to spawn new processes and execute arbitrary commands were denied, restricting the service to its original execution context.
- **File System Abuse** (`ATTACK_FILE_READ`, `ATTACK_FILE_WRITE`): Access to files outside the allowlist was blocked, preventing unauthorized data leakage or modification.
- **Credential Manipulation** (`ATTACK_CRED`): Operations modifying process credentials (e.g., `setuid`) were denied, effectively preventing privilege escalation.
- **Network Misuse** (`ATTACK_NET`): Only explicitly permitted socket operations were allowed; arbitrary outbound connections were successfully blocked.
- **IPC Abuse** (`ATTACK_IPC`): Inter-process communication mechanisms, such as `ptrace` and shared memory, were disabled to prevent unauthorized interference.

TABLE I
FILE OBJECT BOUNDARIES FOR CASPER SERVICES

Service	Allowed File Paths
dns	`/etc/nsswitch.conf`, `/etc/hosts`, `/etc/resolv.conf`, `/etc/services`
net	`/etc/nsswitch.conf`, `/etc/hosts`, `/etc/resolv.conf`, `/etc/services`
grp	`/etc/nsswitch.conf`, `/etc/group`, `/var/db/cache/group.cache`
netdb	`/etc/nsswitch.conf`, `/etc/protocols`
pwd	`/etc/nsswitch.conf`, `/etc/pwd.db`, `/etc/spwd.db`
sysctl	`/etc/pwd.db`
syslog	`/var/run/log`, `/var/run/logpriv`, `/etc/localtime`, `/etc/pwd.db`, `/dev/console`

Note: Multiple paths are separated by commas. These files are labeled during the boot process using the `rc.d` script.

TABLE II
ATTACK HARNESS MAPPING

Category	Case	Syscall / Action	Kernel Object
Process control	`ATTACK_EXEC`	`execve`	proc / vnode
File system abuse	`ATTACK_FILE_READ`	`open` (read-only)	vnode
File system abuse	`ATTACK_FILE_WRITE`	`open` (write)	vnode
Credential access	`ATTACK_CRED`	`getuid` / `setuid`	cred
Network misuse	`ATTACK_NET`	`socket` / `connect`	socket
IPC abuse	`ATTACK_IPC`	`ptrace` / `shm_open`	proc / ipc
Kernel primitives	`ATTACK_KLD`	`kldload`	kernel
Kernel primitives	`ATTACK_SYSCTL`	`sysctlbyname`	kernel

- **Kernel Primitives** (`ATTACK_KLD`, `ATTACK_SYSCTL`): Direct interaction with the kernel, including module loading and system parameter modification, was rejected.

In all cases, enforcement occurred at the kernel object level. This demonstrates that even under full compromise, the attacker remains confined within the strict security boundaries defined by our policy.

B. Performance Evaluation

1) Experimental Setup: We conduct our experiments on both AMD64 and ARM64 platforms:

- Raspberry Pi 4 Model B: 8 GB RAM, ARM Cortex-A72 CPU.
- AMD64-based Machine: AMD Ryzen 5 5600G with Radeon Graphics, 12 logical cores, 16 GB RAM.

All experiments are performed on `FreeBSD 15` with a NO-DEBUG kernel. We disable SMP and unused drivers, compile with `-O0`, and use dynamic linking for testing.

2) Measurement Methodology: We evaluate our method across multiple Casper services. Each service provides different functions, and we measure the execution time of these functions with and without our proposed module.

To ensure robustness, we report the mode of execution times derived via Kernel Density Estimation (KDE) across multiple iterations, thereby mitigating the impact of outliers.

For network-related services, we use a local DNS server to reduce the impact of network latency.

C. System Call Overhead

Figure 3 summarizes the syscall overhead on both ARM64 and AMD64 under three configurations: (1) baseline without our MAC module, (2) MAC module loaded but no labels applied, and (3) MAC module loaded with different labels.

We focus on syscalls that are exercised by our MAC module and are relevant to Casper services. Overall, both architectures exhibit similar trends.

a) Open/close: This microbenchmark measures the overhead of opening and closing a file once. With our MAC module enabled, the measured time increases from 0.993 to 1.050 on ARM64, and from 1.152 to 1.214 on AMD64. The additional overhead is 5.74% on ARM64 and 5.38% on AMD64.

For `dns`, `net` and `fileargs`, the "labeled" configuration shows similar results to the "no-label" case, indicating that label checks add little extra cost beyond the module itself. For other services, attempts to open forbidden files (e.g., `/e`

`tc/hosts`) are denied by our MAC policy, which can reduce the measured time because the syscall fails early.

b) Socket create/close: This microbenchmark measures the overhead of creating and closing an `AF_INET` socket. On ARM64, the measured time increases from 1.265 to 1.304, corresponding to an additional overhead of 3.08%. On AMD64, the measured time changes from 1.011 to 1.077, corresponding to 6.5%.

Among the evaluated services, `dns` and `net` is allowed to create `AF_INET` sockets; therefore, its labeled and no-label configurations show similar trends. Other services are denied by our MAC policy, since they are not expected to perform network socket operations.

c) Sysctl: This microbenchmark measures the overhead of the `sysctl` syscall under different MAC configurations. On ARM64, the measured time increases from 1.235 to 1.385, which corresponds to an overhead of 12.15%. On AMD64, the measured time increases from 1.14 to 1.313, which corresponds to an overhead of 15.17%.

For services that legitimately use `sysctl` (e.g., `dns`, `net`, `sysctl`, and `syslog`), the labeled configuration remains close to the no-label case, suggesting that label checks contribute minimal additional overhead. Other services are denied by our MAC policy when attempting disallowed `sysctl` operations.

d) Overall: Across the evaluated syscalls, enabling our MAC module introduces modest overhead (typically within single-digit percentages), while enforcing least-privilege constraints on Casper services.

D. Function-Level Performance

We report the performance results for each service and function. The results are presented as relative changes compared to the baseline (without our module).

Figure 4 and Figure 5 show the performance comparison of our method on ARM64 and AMD64, respectively. Overall, the overhead remains within 6% for most functions on both platforms, with one function reaching approximately 8%.

To further evaluate the impact on network services, we measure query throughput in Queries Per Second (QPS). Figure 6 shows the QPS for the DNS service on both ARM64 and AMD64. The test was conducted for 60 seconds, and we report the modal QPS across runs for each function. As shown, DNS query throughput decreases by approximately 2–4% on ARM64 and less than 3% on AMD64; the overall trend across functions remains consistent, indicating modest overhead without altering service behavior.

Overall, these results indicate that our method achieves a good balance between security and performance. Both CPU-bound and network-bound workloads confirm the practical applicability of our module in real-world scenarios.

VI. Summary

We present a MAC-based sandboxing approach for Casper services in FreeBSD, addressing security gaps inherent to Casper's normal-mode execution. Our method preserves Capsicum's capability-based guarantees while enabling secure delegation of network and system operations. The performance overhead is minimal, making this a practical enhancement for FreeBSD security.

VII. Future Work

The solution can be extended to support other libraries. For example, the Name Service Switch (NSS) library also needs to access files such as `/etc/hosts` and `/etc/passwd`. We can analyze the NSS library and derive a least-privilege policy that allows access only to the required files.

We can also generalize the module by defining a set of APIs. If a library implements these APIs, the module can enforce restrictions accordingly. For example, if a library implements a file-open API, the module can restrict which files it is allowed to open.

References

[1] M. Zaborski, "Capsicum and casper: A fairy tale about solving security problems." Presentation at AsiaBSDCon 2016, Tokyo, Japan, 2016. URL: https://papers.freebsd.org/2016/asiabsdcon/oshogbo-capsicum_and_casper/.

[2] R. N. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical capabilities for {UNIX}," in *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[3] I. Goldberg, D. Wagner, R. Thomas, E. A. Brewer, *et al.*, "A secure environment for untrusted helper applications: Confining the wily hacker," in *Proceedings of the 1996 USENIX Security Symposium*, vol. 19, USENIX Association Berkeley, 1996.

[4] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[5] B. W. Lampson, "Protection," *SIGOPS Oper. Syst. Rev.*, vol. 8, p. 18–24, Jan. 1974.

[6] M. Kerrisk, "Using seccomp to limit the kernel attack surface." https://man7.org/conf/ndctechtown2018/limiting-the-kernel-attack-surface-with-seccomp-NDC-TechTown-Kerrisk.pdf, 2018. Accessed: 2025-09-27.

[7] J. Corbet, "A seccomp overview." https://lwn.net/Articles/656307/, 2015. Accessed: 2025-09-27.

[8] B. Beck and D. Mazières, "Pledge and unveil in openbsd." https://www.openbsd.org/papers/BeckPledgeUnveilBSDCan2018.pdf, 2018. Accessed: 2025-09-27.

[9] LWN.net, "Openbsd's unveil()." https://lwn.net/Articles/767137/, 2018. Accessed: 2025-09-27.

[10] S. Topics, "Mandatory access control and selinux." https://www.sciencedirect.com/topics/computer-science/mandatory-access-control, 2020. Accessed: 2025-09-27.

[11] A. Project, "Apparmor linux security module." https://wiki.ubuntu.com/AppArmor, 2005. Accessed: 2025-09-27.

[12] R. N. Watson, "Adding trusted operating system features to freebsd," in *USENIX Technical Conference, Boston, MA*, 2001.

[13] R. N. M. Watson, *New Approaches to Operating System Security Extensibility*. PhD thesis, University of Cambridge, Cambridge, UK, 2012. UCAM-CL-TR-818.

[14] B. Gregg and J. Mauro, *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.

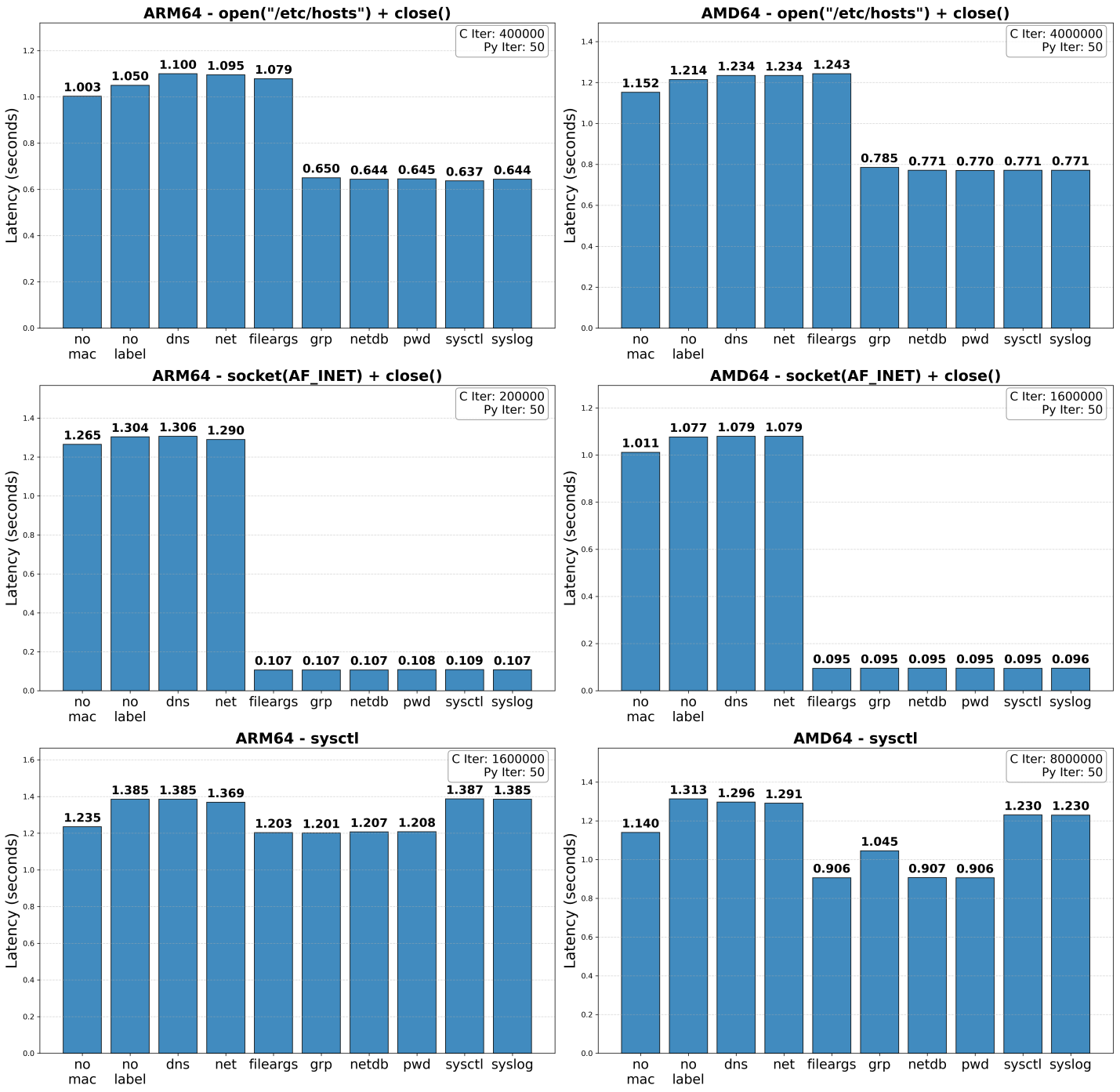

Fig. 3. Syscall Overhead under ARM64 and AMD64

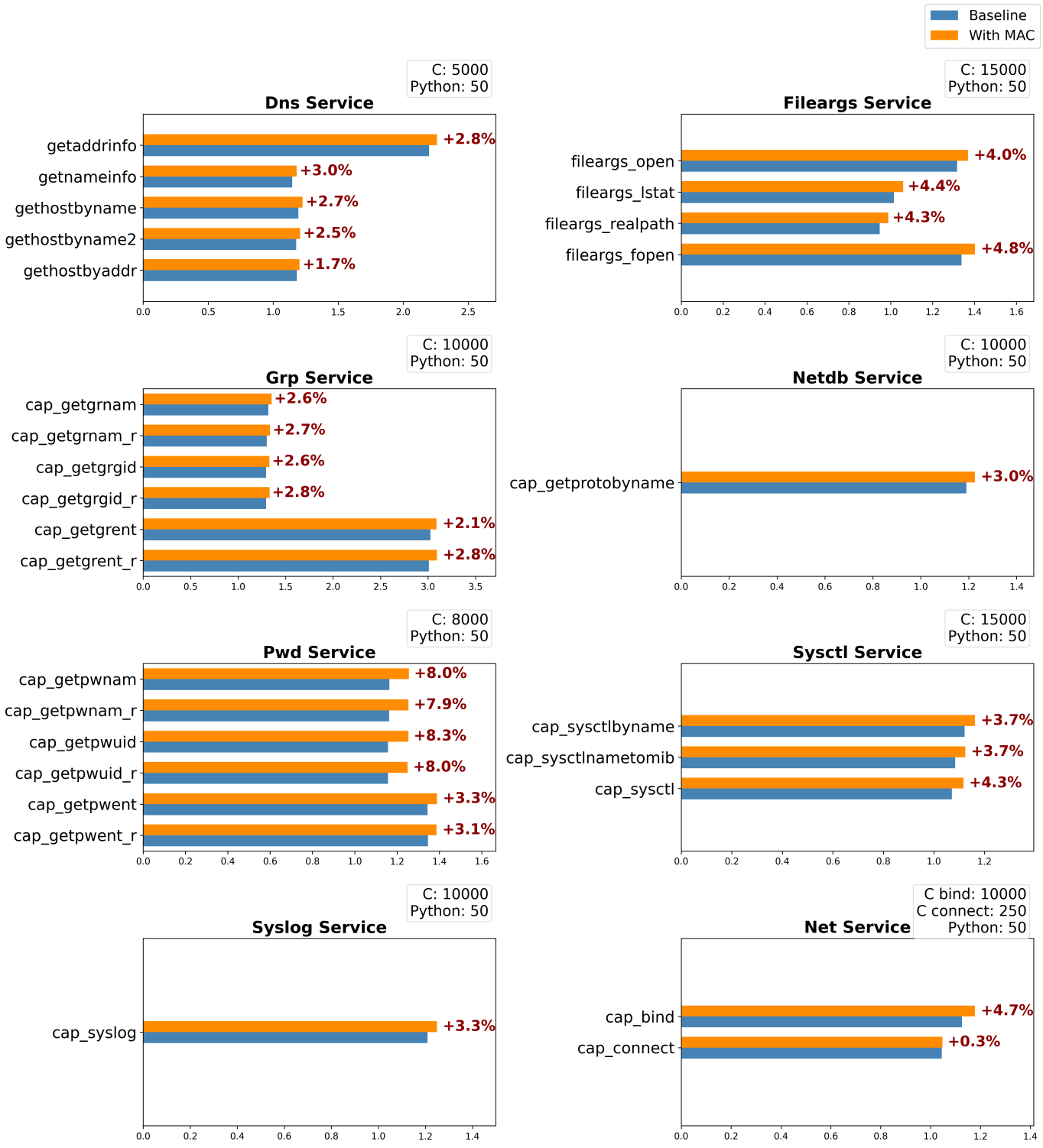

Fig. 4. Performance on ARM64

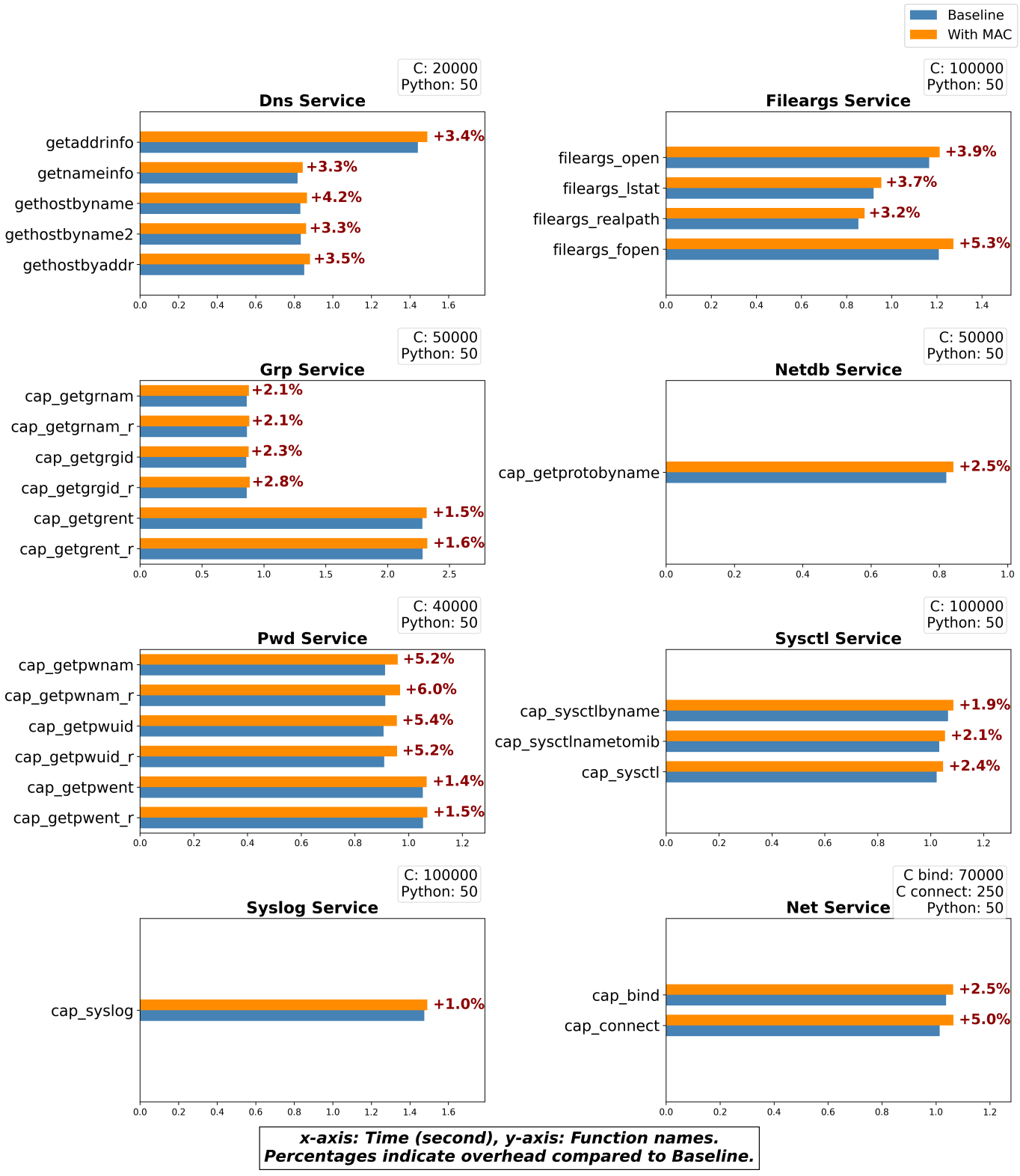

Fig. 5. Performance on AMD64

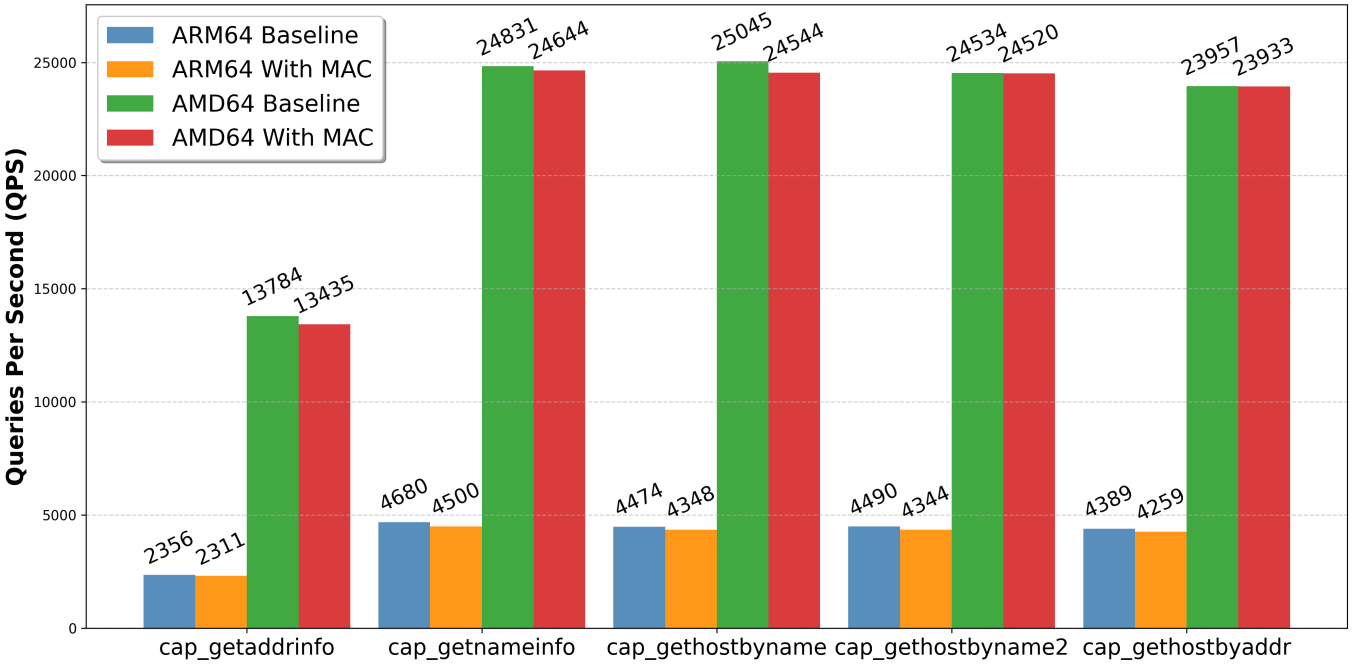

Fig. 6. QPS under ARM64 and AMD64